实验5 存储器设计实验

一、实验目的

- 学习使用Vivado或者ISE开发工具的Memory IP核，设计生成存储器模块的方法；
- 学习存储器的结构及读写原理，掌握存储器的设计方法。

二、实验内容与原理

- **实验内容：**
  - 使用Vivado或者ISE开发工具，用IP核导航设计一个64×32位的存储器IP核；
  - 引用该IP核构造一个存储器实例，实现存储器模块。

1、RISC-V的存储器

2、使用Vivado的IP核构造存储器模块

3、使用ISE的IP核构造存储器模块

1、RISC-V的存储器

- （1）32位RISC-V处理器的存储器特征
  - 地址总线：32位
  - 数据总线：32位
  - 编址方式：按字节编址
  - 存储器容量：2^{32}×8位=4GB
  - 地址范围：0x00000000~0xFFFFFFFF
  - 多字节存储：
    - 指令：以小端模式存储在程序存储器；
    - 数据：原先只允许小端模式（little-endian），V2.1版本也允许大端模式（big-endian）或者大小端交替模式

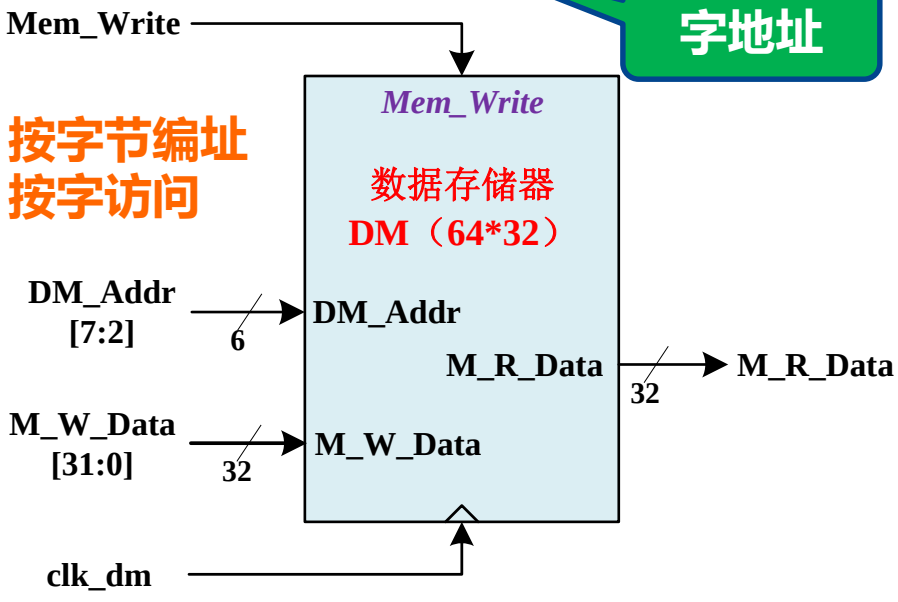
1、RISC-V的存储器

字节地址

- （2）本实验实现的RV32I处理器的数据存储器
  - 单端口RAM
  - 具有读写功能
  - 小端模式
  - 逻辑上：256×8位，地址8位，数据8位
  - 物理上：64×32位，地址6位，数据32位

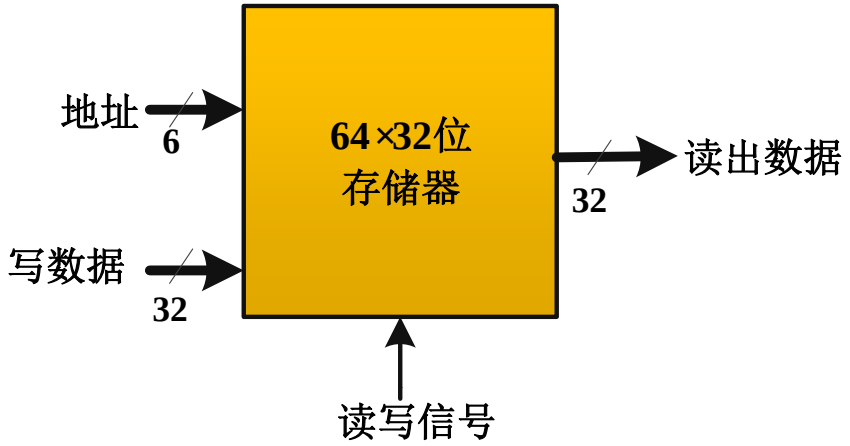
字地址

字地址	MSB 31 24	23 16	15 8	LSB 7 0
	位序			
0	3	2	1	0
4	7	6	5	4
…	…	…	…	…
56	59	58	57	56
60	63	62	61	60



1、RISC-V的存储器

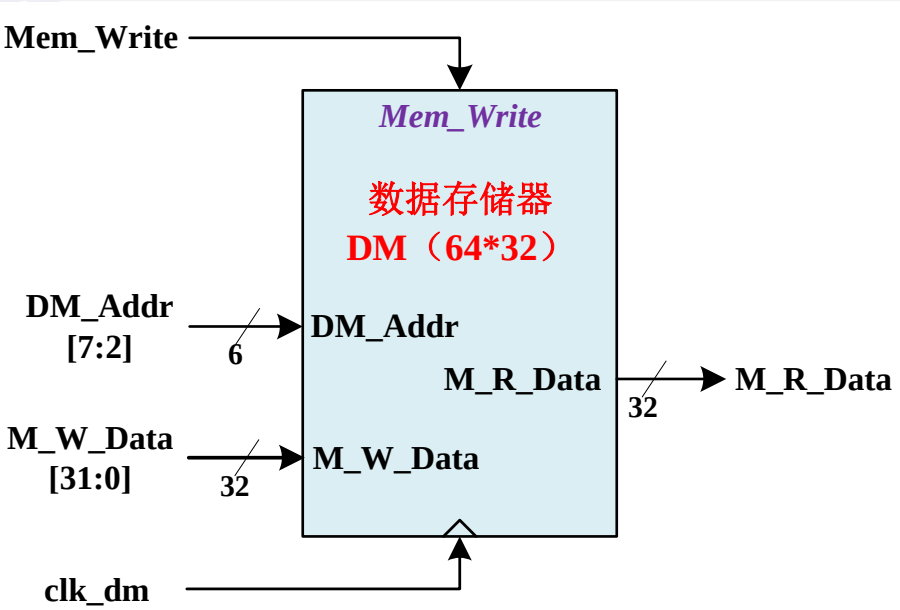
- 单端口RAM：
  - 一组地址：6位
  - 一组读出数据：32位
  - 一组写入数据：32位
  - 读写信号



- 本实验的存储器：
  - 访问方式：按字（32位）访问
  - 组织方式：4个字节/字，即64×32位
  - 即：按字节编址，按字访问（边界对齐：A_1A_0=00）

1、RISC-V的存储

- 在时钟clk_dm上跳沿进行读写操作
  - Mem_Write=0：读操作
  - Mem_Write=1：写操作

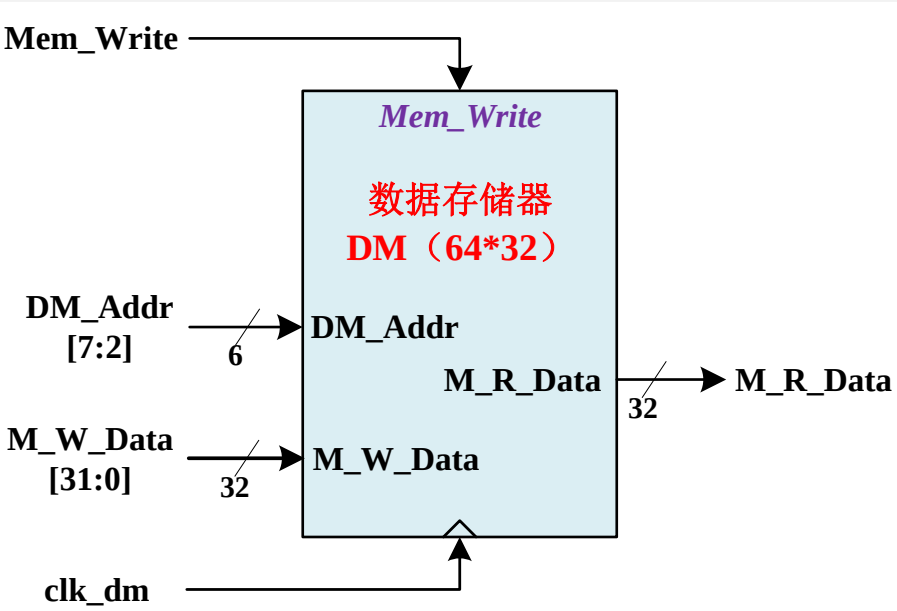


功能表

clk_dm	Mem_Write	地址输入	数据输入	数据输出	操作
↑	0	DM_Addr	——	M_R_Data	读操作
↑	1	DM_Addr	M_W_Data	——	写操作
其他	——	——	——	——	无操作

1、RISC-V的存储

- 本次实验：给出的8位（后续为32位地址），只使用[7:2]6根地址，可忽略最低两位地址。
  - 因为至少要实现lw和sw指令



- 边界对齐（地址对齐）
  - 字访问：$A_1A_0=00$ （4的整数倍）
  - 半字访问：$A_0=0$ （2的整数倍）
  - 双字访问：$A_2A_1A_0=000$ （8的整数倍）

- lw(取字)、sw(存字)
- lh/lhu(取半字)、sh(存半字)
- lb/lbu(取字节)、sb(存字节)
- fld/fsd(取/存浮点双字)

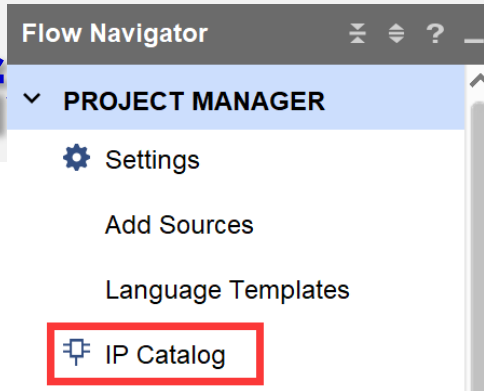
1、RISC-V的存储器

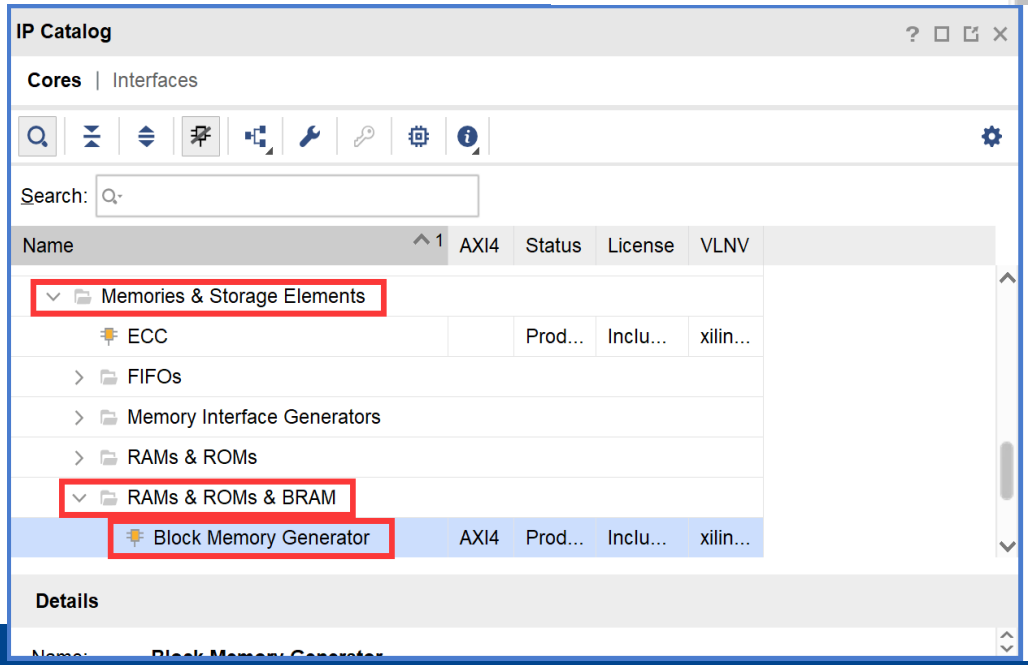
- (3) 实现方法：两种方法构造：
  - 方法一：采用Verilog语言中存储器类型（即reg的数组类型）定义，并自行管理；
    - 模块及其读写操作：需要简单编程来完成
    - 存储器内容的初始化：在模块内编程赋值完成；
  - 方法二：使用EDA软件内置的存储器IP核来实现。
    - 模块及其读写操作：无需编程，通过向导生成存储器模块，然后引用其实例。
    - 存储器内容的初始化：可以和外部的一个格式化文件关联，生成存储器模块时，自动从该文件装载程序或数据。

性能更好、更可靠

2、使用Vivado的IP核构造存储

- （1）选择Memory IP核
  - ① 打开或新建一个Vivado工程，Flow Navigator→PROJECT MANAGER→IP Catalog→单击
  - ② 选择Memories & Storage Elements→RAMs & ROMs & BRAM→Block Memory Generator→双击



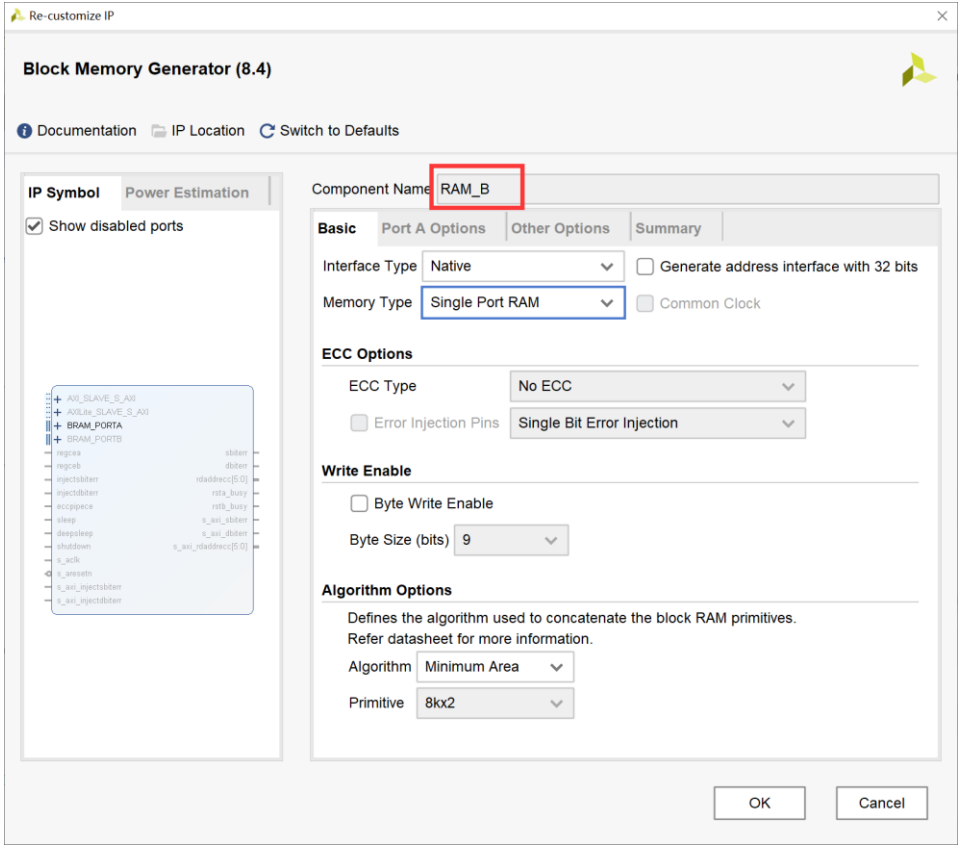


2、使用Vivado的IP核构造存储器模块

- （2）设置Memory IP核的参数

① 输入部件名称“Component Name”为：RAM_B

② 第一标签页Basic：存储器类型“Memory Type”选择单端口RAM“Single Port RAM”

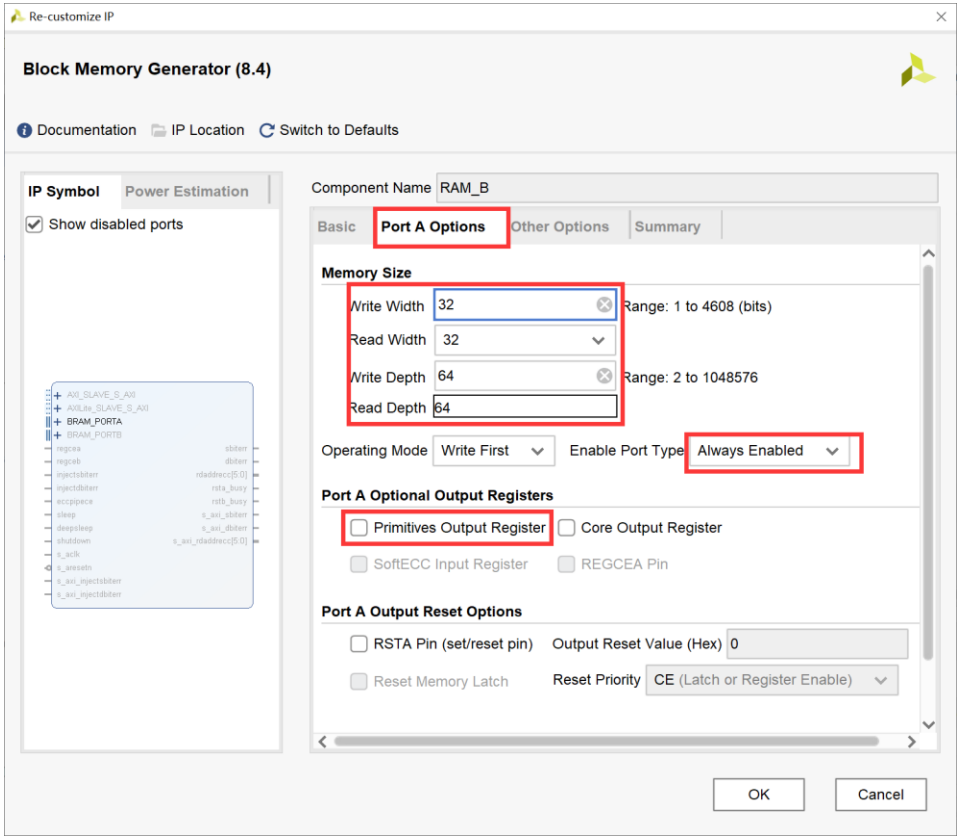


2、使用Vivado的IP核构造存储器模块

- （2）设置Memory IP核的参数

③ 第二标签页Port A Options：

- 设置存储器的读写宽度Width：存储单元字长，32位
- 读写深度Depth：存储单元个数，64个
- 端口使能类型Enable Port Type：选择“Always Enabled”
- 不勾选“Primitives Output Register”选项

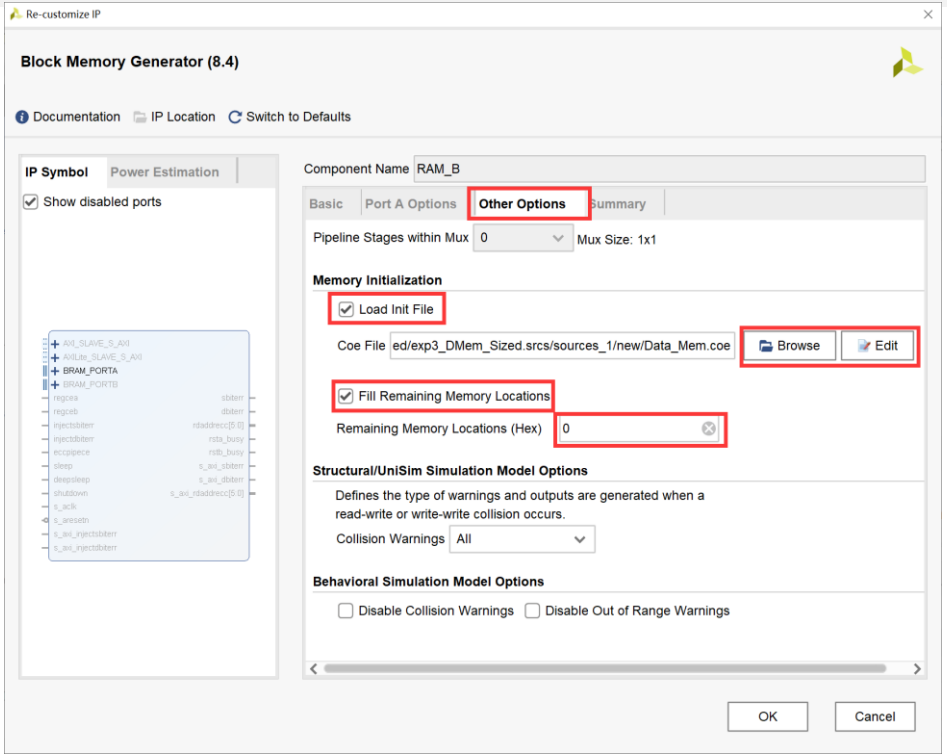


2、使用Vivado的IP核构造存储器模块

- （2）设置Memory IP核的参数

④ 第三标签页Other Options：

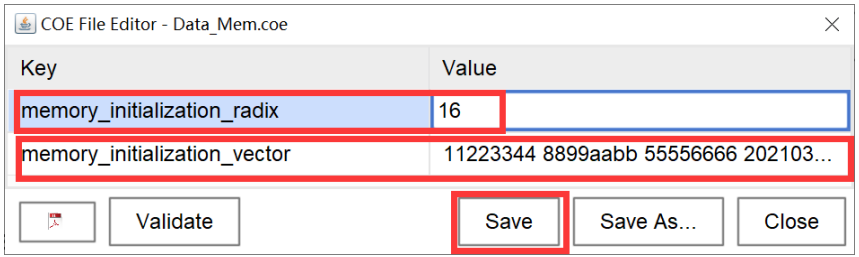
- 选中Load Init File选项；
- 点击Browse按钮，则可以选择一个已经生成的COE文档（*.coe）作为关联初始化参数；
- 点击Edit按钮，则可以新建一个COE文档，并弹出编辑窗口，可直接输入COE文件的内容



- 勾选“Full Remaining Memory Locations”，并在下方的编辑框内输入一个16进制数值，用它填满存储器的其他单元

2、使用Vivado的IP核构造存储器模块

- （2）设置Memory IP核的参数



④ 第三标签页Other Options：

- COE关联文件
  - 后缀名为*.coe，纯文本文件
  - 可以用任意纯文本编辑器（记事本、写字板、gedit等等）创建和编辑
  - 文件内容格式：

关键字　　data采用的进制　　初始化数据向量

```
memory_initialization_radix=<Radix>;
memory_initialization_vector=<data1>, <data2>, ... <data n>;
```

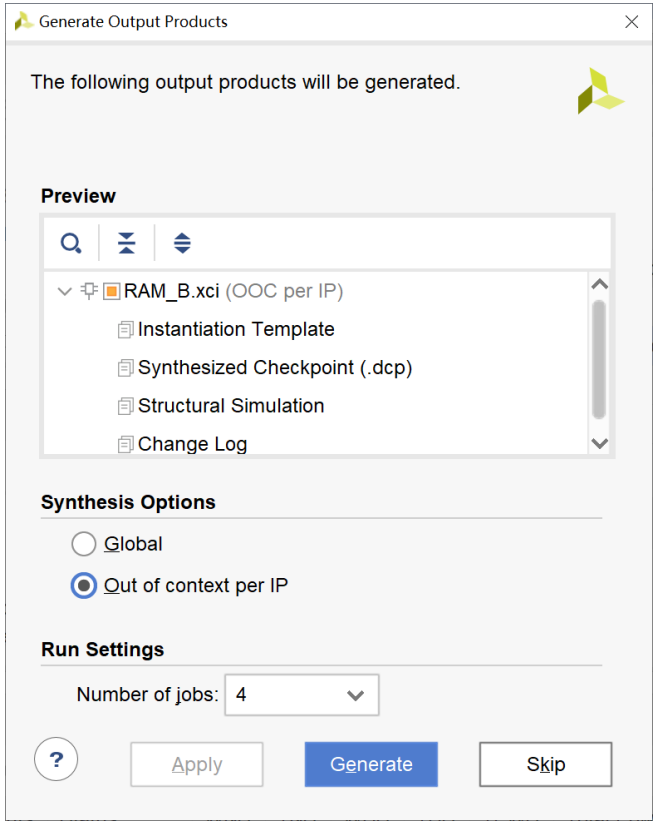
- 可在IP核向导的编辑窗口中编辑、创建

2、使用Vivado的IP核构造存储器模块

- （2）设置Memory IP核的参数

⑤ 第四标签页Summary：无需设置

- 点击“OK”按钮→对话框
- 单击“Generate”，即可生成



- 每当*.coe文件内容修改后，都需要重新打开IP核，点击“OK”和“Generate”执行生成操作，才能根据更新内容重新初始化存储器。
- 双击RAM_B，就可以再次启动Block Memory Generator向导。

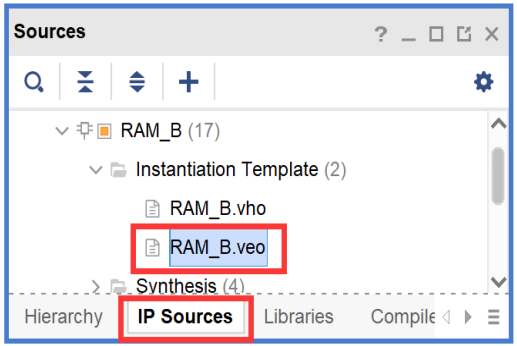
2、使用Vivado的IP核构造存储器模块

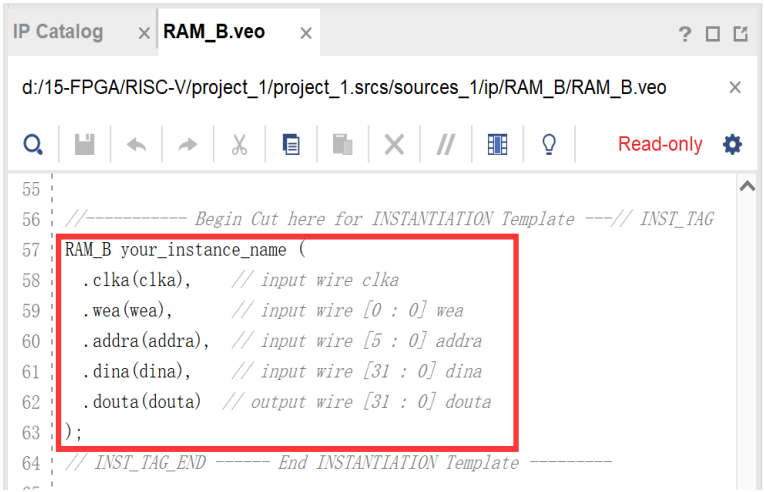
（3）引用存储器IP核生成实例

① Source窗口→单击下方的“IP Source”标签页→Instantiation Template

② 双击Verilog的模板文件“RAM_B.veo”，拷贝引用示例代码到存储器模块

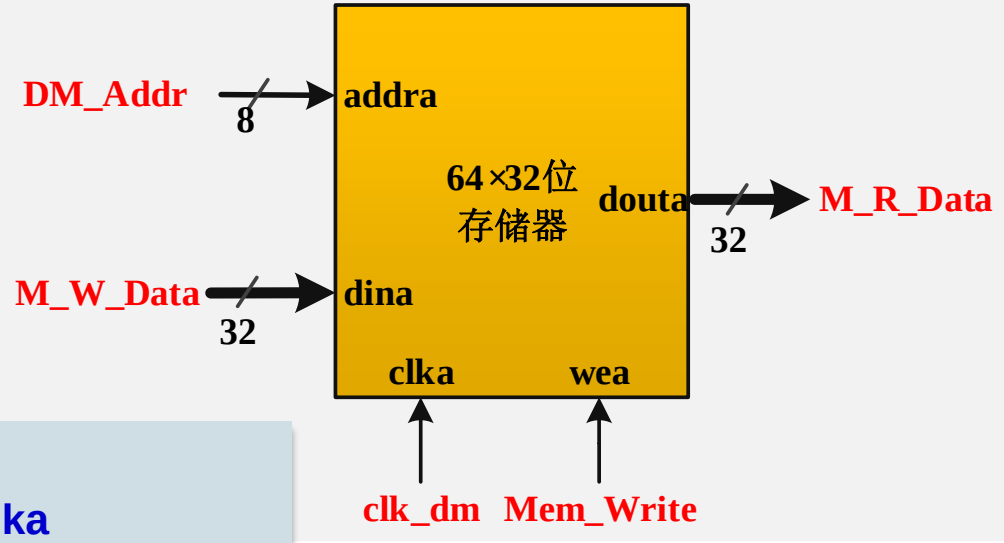
③ 修改实例名，定义存储器模块的输入输出端口，与IP核实例相连





2、使用Vivado的IP核构造存储器模块

- （3）引用存储器IP核生成实例




```
RAM_B    Data_RAM (
  .clka(clk_dm),                // input clka
  .wea(Mem_Write),              // input [0 : 0] wea
  .addra(DM_Addr[7:2]),         // input [5 : 0] addra
  .dina(M_W_Data),              // input [31 : 0] dina
  .douta(M_R_Data)              // output [31 : 0] douta
);
```

3、使用ISE的IP核构造存储器模块

- ①在工程目录下，新建并编辑一个关联文档。
  - 关联文件：后缀名为*.coe，纯文本文件
  - 文件内容格式：

关键字

数据格式：data采用的进制

初始化数据向量

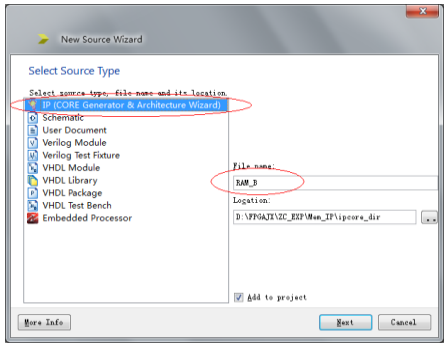
```
memory_initialization_radix=<Radix>;
memory_initialization_vector=<data1>, <data2>, ... <data n>;
```

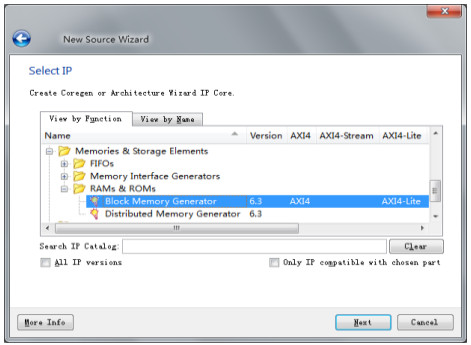
- 例：初始化存储器的前10个单元，地址从0开始；
- 数据在第二行“=”右边，16进制表示，以逗号分隔、以分号结束。

```
memory_initialization_radix=16;
memory_initialization_vector=00000820,00632020,00010fff,20006789,FFFF0000,0000
FFFF,88888888,99999999,aaaaaaaa,bbbbbbbb;
```

3、使用ISE的IP核构造存储器模块

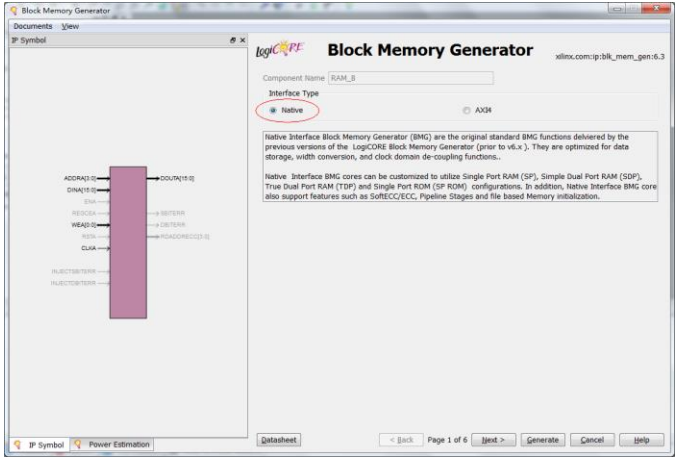
- ②新建一个Memory IP核。
  - 选择Project→New Source,弹出New Source Wizard对话框。
  - 选择IP（Core Generator & Architecture Wizard），输入存储器IP核名称；存放路径为工程目录的子目录\ipcore_dir下。
  - 点击Next进入IP核选项，选择Memories & Storage Elements→RAMS & ROMs→Block Memory Generator，之后点击Next,再点击Finish进入Memory参数设置。

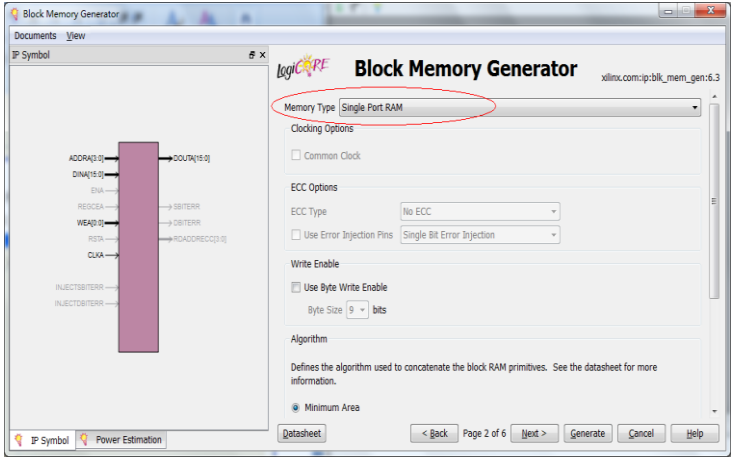




3、使用ISE的IP核构造存储器模块

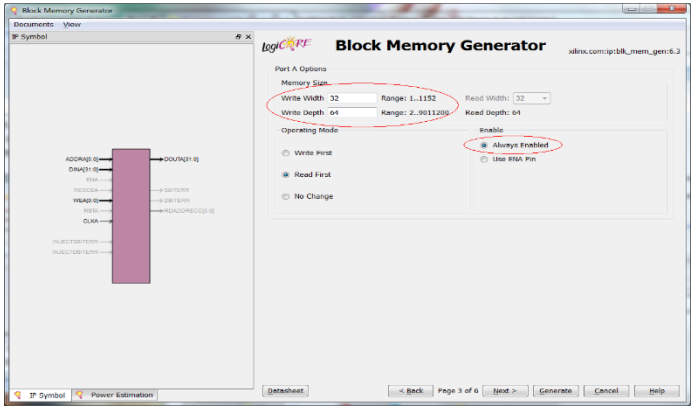
- ③设置Memory IP核的参数。
  - 默认选择接口类型Interface Type为：Native，点击Next进入第2页设置存储器类型。
  - 只修改Memory Type选择项为：Single PortRAM,点击Next进入第3页设置端口参数。

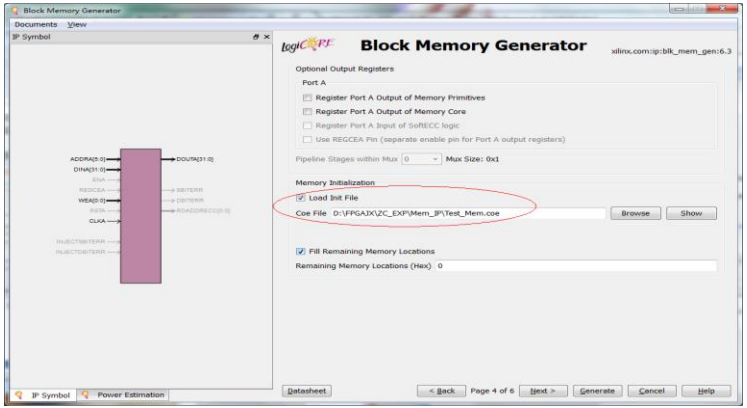




3、使用ISE的IP核构造存储器模块

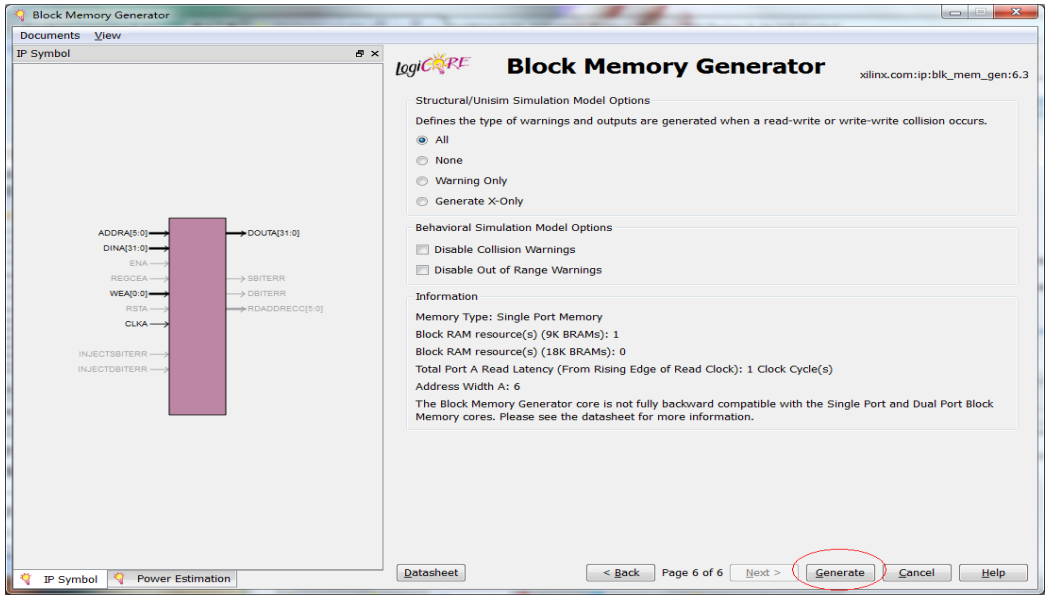
- ③设置Memory IP核的参数。
  - 修改存储器尺寸Write Width 为32、Write Depth 为64，Enable选择Always Enabled,点击Next进入第4页设置Memory初始关联文档。
  - 选中Load Init File选项，点击Browse按钮，选择第一步生产的COE文档作为关联初始化参数，点击Next进入下一页。





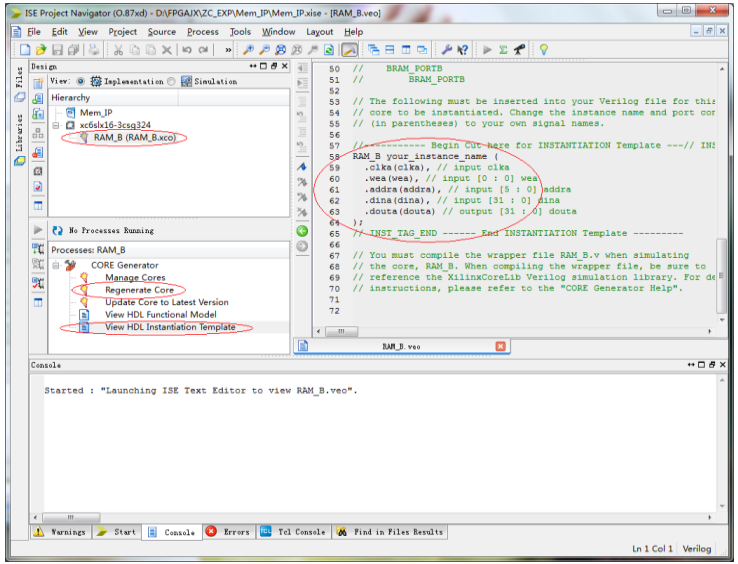
3、使用ISE的IP核构造存储器模块

- ③设置Memory IP核的参数。
  - 之后，无需修改，点击Next进入下一页，再点击Generate按钮，系统自动生成RAM_B存储器模块。



3、使用ISE的IP核构造存储器模块

- ④在上层模块中，调用该存储器模块。
  - 双击过程管理区的View HDL Instruction Template，在右侧会给出RAM_B的调用模板，将其拷贝到上层模块中，修改实例名称和连接端口参数即可引用。



3、使用ISE的IP核构造存储器模块

- ①在工程目录下，新建并编辑一个关联文档。
- ②新建一个Memory IP核。
- ③设置Memory IP核的参数。
- ④在上层模块中，调用该存储器模块。

读写时钟

写使能

地址

写入数据

```
RAM_B your_instance_name (
  .clka(clka),      // input clka
  .wea(wea),        // input [0 : 0] wea
  .addra(addra),    // input [5 : 0] addra
  .dina(dina),      // input [31 : 0] dina
  .douta(douta)     // output [31 : 0] douta
);
```

读出数据

- 读时序：
  - addra=单元地址，wea=0
  - clka上跳沿（douta=读出数据）
- 写时序：
  - addra=单元地址， dina=写入数据，wea=1
  - clka上跳沿

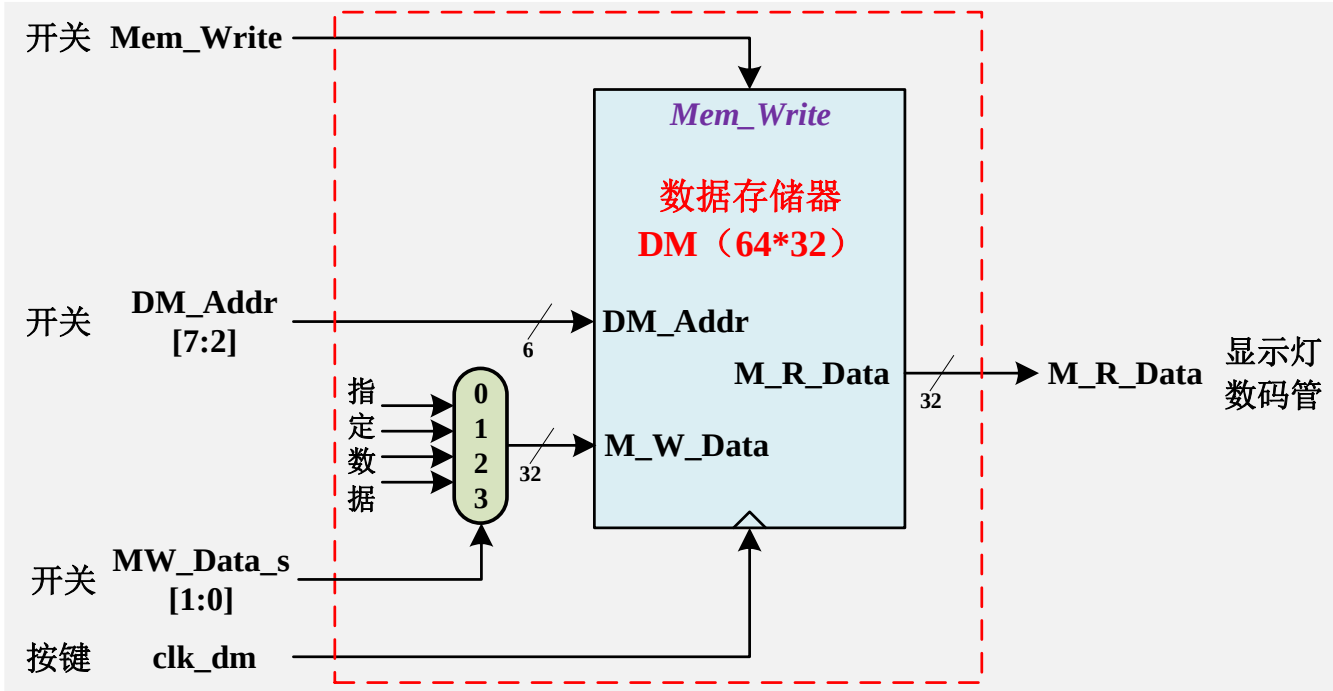
三、实验要求

1. 使用Vivado或者ISE工具，按照上述步骤新建一个64×32位的Single Port RAM，并使用COE关联文件初始化其内容。
2. 新建一个存储器模块，引用上述的存储器IP核，连接实例与存储器模块的端口。
3. 编写仿真激励代码，进行仿真测试，确保存储器模块读写逻辑功能正确。

三、实验要求

4. 针对使用的实验板卡，设计存储器模块的**板级验证**实验方案，编写**顶层测试模块**。

- 写入数据：指定几组常数



三、实验要求

5. 选择存储器单元地址，执行读写操作，验证你的存储器是否能正常存取，将实验结果记录到表格中，要求存储器读和写功能都被有效测试。

存储器地址	初始化数据	读出数据	写入新数据	读出数据

三、实验要求

6. 撰写实验报告（不做要求），格式见附录，重点内容包括：
   - 对仿真结果进行分析；
   - 描述你设计的板级验证实验方案、模块结构与连接；
   - 说明你的板级操作过程；
   - 分析记录下来的板级实验结果、得到有效结论。
   - 请力所能及回答或实践本实验的“思考与探索”部分。

四、实验步骤

1. 新建一个工程，通过IP核生成向导创建一个64×32位的单端口RAM IP核，并创建与编辑一个COE文件，初始化该RAM的内容。
2. 新建一个存储器模块，其中引用上述生成的RAM IP核，并正确连接存储器模块端口与IP核实例的端口。
3. 编写激励代码，仿真验证存储器模块功能；分析仿真结果，确保读写操作正确。
4. 设计存储器模块板级验证的实验方案，然后据此编写一个顶层测试模块。
5. 可以依据实际需要，对顶层测试模块进行仿真测试，或者直接进入管脚约束环节。

四、实验步骤

6. 新建管脚约束文件，依据提示和设计的板级验证实验方案，进行相应的引脚配置。
7. 生成*.bit文件，下载到实验设备的FPGA芯片中。
8. 板级实验：按照你所设计的实验方案，操作输入设备、观察输出设备

存储器读操作

① 拨动开关输入存储器单元地址
② 拨Mem_Write开关=0，按时钟键clk_dm
③ 观察输出设备上读出的存储器数据是否正确（与初始化数据一致，或者与最后一次写入的数据一致）；

存储器写操作

① 拨动开关输入存储器地址；
② 拨动开关，选择或者输入准备写入存储器的数据；
③ 拨Mem_Write开关=1，按动时钟键clk_dm

要确认：写入存储器的数据已经更新了指定存储器单元的值

五、思考与探索（至少完成1道）

1. 在仿真测试和板级测试中，你是如何确认存储器单元被写入成功的？请具体说明。
2. 在板级实验中，你执行存储器写操作时，按下clk_dm时钟键后，观察存储器读出的数据，是新写入的数据还是原来的数据？请你分析为什么会是这样的现象。
3. 接上题，请通过仿真测试，观察写操作的clk_dm来临后，读出的数据是什么？和板级验证结果一致吗？分析原因。
4. 在存储器IP核生成向导中，本实验选择的Memory Type是Single Port RAM，尝试选择其他类型，简单比较一下生成的存储器IP核的端口各自有什么不同。

五、思考与探索（至少完成1道）

5. 本实验实现的32位RISC-V的存储器，虽然是按照字节编址的，但是按照字（32位）访问，物理存储器也是按照字来组织。我们知道RV32I的指令集中，还有按照字节（8位）和半字（16位）访问的访存指令，考虑如何实现存储器按照字节和半字访问呢?

- 可添加输入端口Size_s[1:0]：指出访问的尺寸
  - =00：按字节访问
  - =01：按半字访问，忽略DM_Addr[0]
  - =1x：按字访问，忽略DM_Addr[1:0]
- DM_Addr[7:2]：字地址
- DM_Addr[1:0]：字节地址
- DM_Addr[1]：半字地址

五、思考与探索（至少完成1道）

5. 如何实现存储器按照字节和半字访问呢?

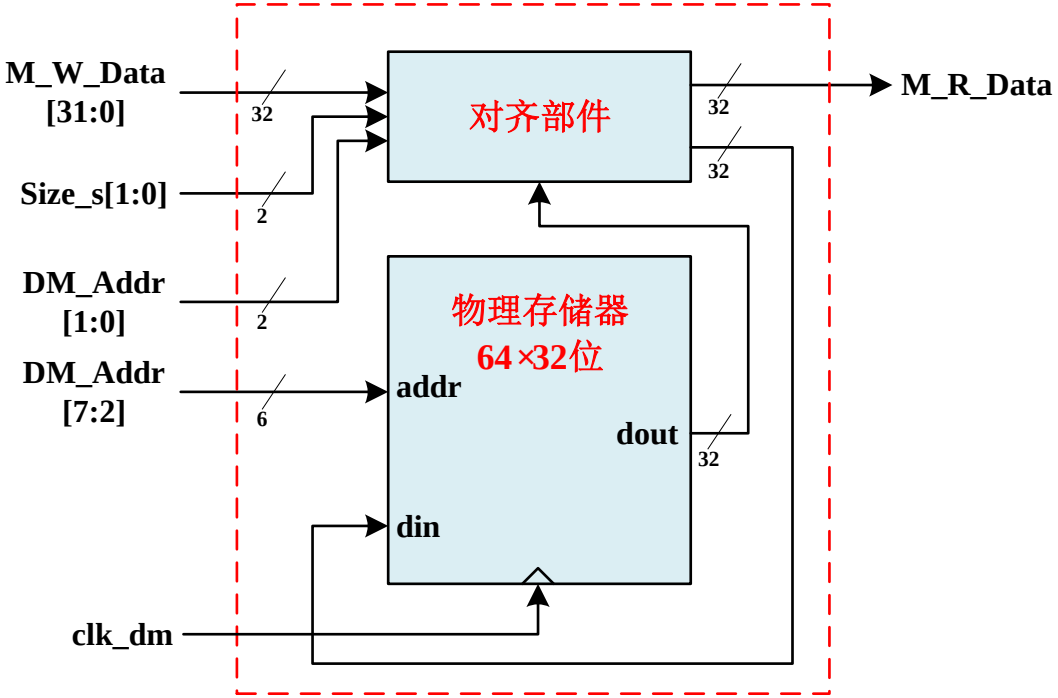
- 假设：Mem[0]=11223344H，写入数据M_W_Data=55667788H
- 则：存储器在不同访问尺寸下的读写0号单元的结果：

访问尺寸Size_s[1:0]	存储器地址 DM_Addr[7:0]	读出数据 M_R_Data[31:0]	写入数据 din[31:0]
=00，按字节访问	=8’ b0000_0000	32’ h0000_0044	32’ h1122_3388
	=8’ b0000_0001	32’ h0000_0033	32’ h1122_7744
	=8’ b0000_0010	32’ h0000_0022	32’ h1166_3344
	=8’ b0000_0011	32’ h0000_0011	32’ h5522_3344
=01，按半字访问	=8’ b0000_000x	32’ h0000_3344	32’ h1122_7788
	=8’ b0000_001x	32’ h0000_1122	32’ h5566_3344
=1x，按字访问	=8’ b0000_00xx	32’ h1122_3344	32’ h5566_7788

五、思考与探索（至少完成1道）

5. 如何实现存储器按照字节和半字访问呢?

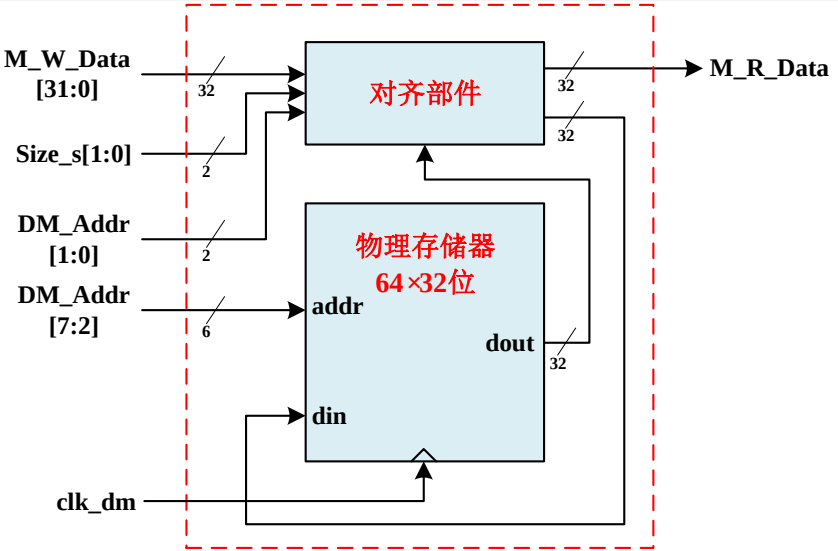
- 对齐部件的实现，可以使用掩码、多路选择器、逻辑运算和移位等多种方式实现



五、思考与探索（至少完

5. 如何实现存储器按照字节和半字访问呢?

- 掩码+移位方式实现读操作:
- shift_n：取决于Mask_Index
  - 读字节：可能为0/8/16/24
  - 读半字：则可能是0/16



写操作呢?

```
assign Mask_Index = {Size_s, DM_Addr[1:0]};
wire [31:0] Mask[0:15]= {
    32'h0000_00ff,32'h0000_ff00,32'h00ff_0000,32'hff00_0000,  //byte size
    32'h0000_ffff,32'h0000_ffff,32'hffff_0000,32'hffff_0000,  //half word size
    32'hffff_ffff,32'hffff_ffff,32'hffff_ffff,32'hffff_ffff,  //word size
    32'hffff_ffff,32'hffff_ffff,32'hffff_ffff,32'hffff_ffff   //word size
};
assign M_R_Data = (dout & Mask[Mask_Index]) >> shift_n;
```